COMX 35/PC-1

ASSEMBLER MANUAL

INTRODUCTION TO ASSEMBLY PROGRAMMING

R & D Dept.,
COMX World Operations ltd.

CHAPTER 1

INTRODUCTION

1.1 What is machine language

Machine language is the lowest level computer language. It is the only kind of language which a Central Processing Unit (CPU) can interpret. Machine language is a series of machine codes. These machine codes develope the motion of a CPU. According to different machines codes, CPU does different kinds of work, such as addition and subtraction. Each CPU has its own group of machines codes. One group of these codes is called an instruction set. The CPU of COMX 35/PC-1 is RCA CDP1802 and the instruction set used is CDP1802 instruction set.

Machine codes are in fact some binary codes having special meaning. Each code stand for a specified CPU motion. By chaining up a sequence of these codes, CPU can process many different kinds of work. Even large and complicated task can be done in this way.

1.2 Why use assembly language

Machine codes are meaningless to human beings. They are only meaningful to CPU. Since memorizing a series of meaningless code is difficult, some simple text is used instead of the machine codes. They are called "mnemonics". Each mnemonic stand for one machine code. From the text of a mnemonic, the meaning of the corresponding code can be estimated. Mnemonics composed a higher level language called "assembly language". Assembly language is more rational and is easier to be understood by human beings. Programs written in assembly language are called "assembly programs".

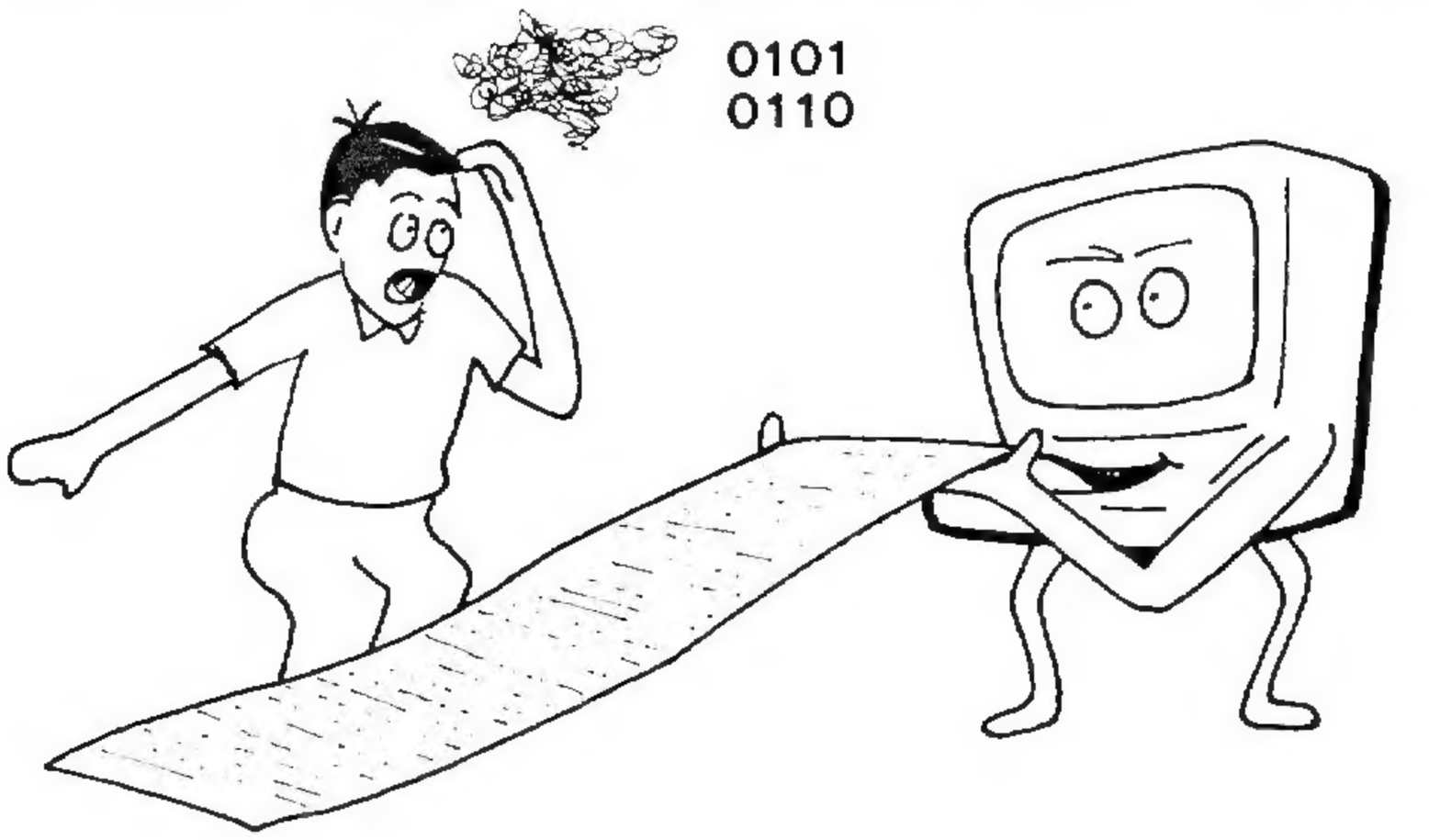

Fig. 1.2 Machine codes are meaningless to human beings.

1.3 Use of assembler

Assembly programs contain a series of mnemonics. These mnemonics must be converted into manchine codes before the program is executed. Assembler is the tool which convert assembly programs into machine codes. Moreover, according to assembler's limitation, some comments, labels and optional commands can be added to assembly programs. These optional materials made assembly programming easier and more systematic.

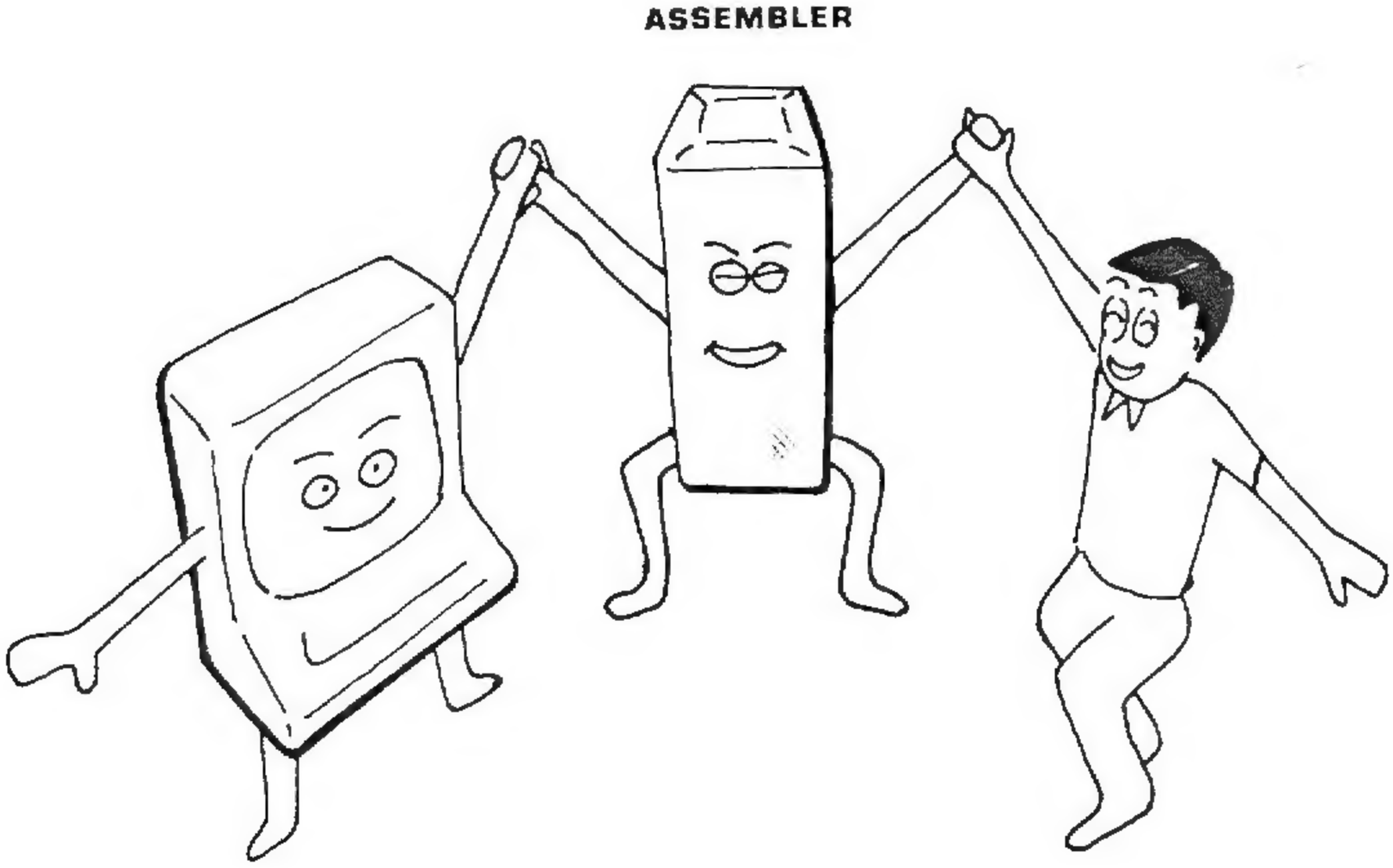

Fig. 1.3 Programming is made easier and more systematic with the help of assembler.

1.4 What happens in COMX 35/PC-1

The assembler of COMX 35/PC-1 is COMX 1802 Assembler. Other than the general features, COMX 1802 Assembler also provide logical and arithmatical operation, pseudo directives and macro instructions. To develop a machine code program, an assembly program should be developed first. This program must follow the COMX 1802 Assembler syntax rules. 1802 Assembler. The program can be written using COMXSTAR, the COMX's word processor. After converted by assembler, a series of machine codes will be generated. They will be saved to disk in text form. This text form machine code file is called HEX file.

In order to execute the program, the HEX file must be converted into binary codes. This job is done by the COMX Loader. COMX Loader load a HEX file and convert it into binary codes. The codes may be filled in RAM or saved to disk. When binary form machine codes are filled in RAM, the program can be executed.

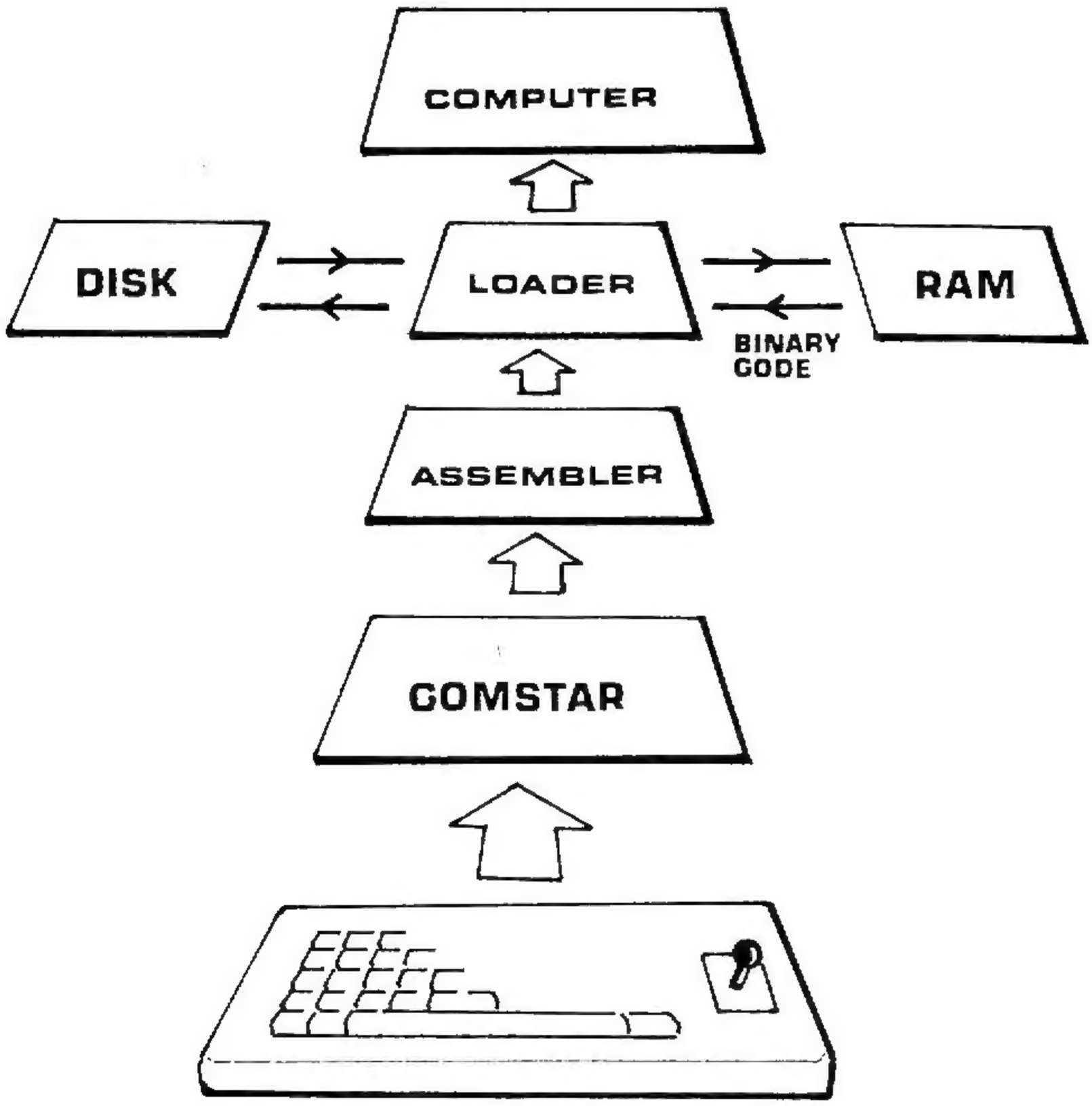

Fig. 1.4 How COMX 35/PC-1 works.

CHAPTER 2

EXAMPLE

Here is an example showing how a machine code program can be developed.

2.1 Source program :

ASSEMBLY LANGUAGE

```
:TWO BYTES SUBTRACTION
:SUBTRACT $1234 FROM $4321
:RESULT PRINT ON SCREEN

::: REGISTER LABEL
SP      EQU 2

::: SUBROUTINE LABEL
OUTPUT EQU 320FH

::: MAIN PROGRAM
        ORG 5000H
START   SEX SP
        LDI 21H
        SMI 34H             :SUBTRACT LOW BYTE
        STXD                :PUSH LOW RESULT
        LDI 43H
        SMBI 12H            :SUBTRACT HIGH BYTE
        CALL OUTPUT         :PRINT RESULT HIGH BYTE
        IRX ; LDX           :PULL LOW RESULT
        CALL OUTPUT         :PRINT RESULT LOW BYTE
        EXIT                :RETURN TO BASIC
        END
```

This program performs a subtraction of two 16 bits numbers, $4321 and $1234. The result will be printed on screen.

2.2 Procedures :

2.2.1 Develope the assemlby program

Key in the above source program using COMXSTAR. Form a text file called "EXAMPLE".

2.2.2 Assemble the assembly program

Use COMX 1802 Assembler to assemble the source file into a HEX file called "EXAMPLE.H".

Steps are shown below :

Note : Words underlined stand for text input.
Words in small letter are explanations.

```
READY
:DOSURUN,"ASM2"                          run COMX 1802 Assembler

ASM1802 VERSION 1.0                      assembler activated

BY COMX WORLD OPERATIONS LTD.

SOURCE FILE (FN. , DR. NO. ; OPTIONS):
>EXAMPLE,1;H L G                         input source file name
                                         select HEX file option
HEX. FILE (FN. , DR. NO.)
>EXAMPLE.H,1                             input HEX file name

DESTINATION:
1 SC, 2 PR, 3 PR & SC, 4 TH, 5 TH & SC
6 DK, 7 DK & SC, 8 NO DEST
>8                                       no Dest file

READY                                    assembling finished
:
```

2.2.3 Form binary codes

Use COMX Loader to convert the HEX file into binary codes.

Steps are shown below :

Note : Words underlined stand for text input.
Words in small letter are explanations.

```
READY
:DOSURUN,"LOADER"                        run COMX Loader

COMX LOADER VERSION 1.0                  COMX Loader activated
COMX WORLD OPERATIONS LTD.,1985 (C)

SOURCE FILE NAME (FN,DR):
>EXAMPLE.H,1                             input HEX file name

SELECTE DESTINATION TYPE:
1.DISK  2.RAM  3.RAM & DISK
>2                                       select fill codes in RAM

READY                                    loading process finished
:
```

The machine code program is now loaded in RAM.

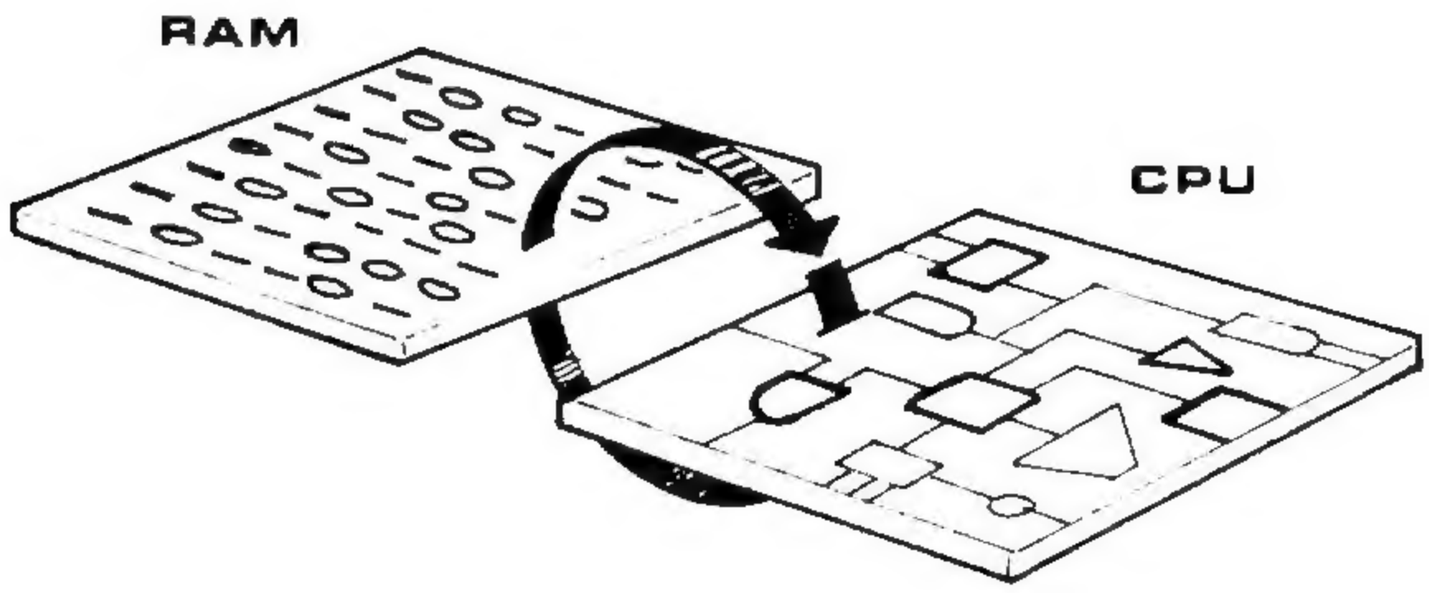

Fig. 2.2 Run the program.

2.3 Run the program

Steps are shown below :

Note : Words underlined stand for text input.
Words in small letter are explanations.

```
READY
:CALL( ■5000)                    run the program
3DED                             program result
READY                            program finished
:
```

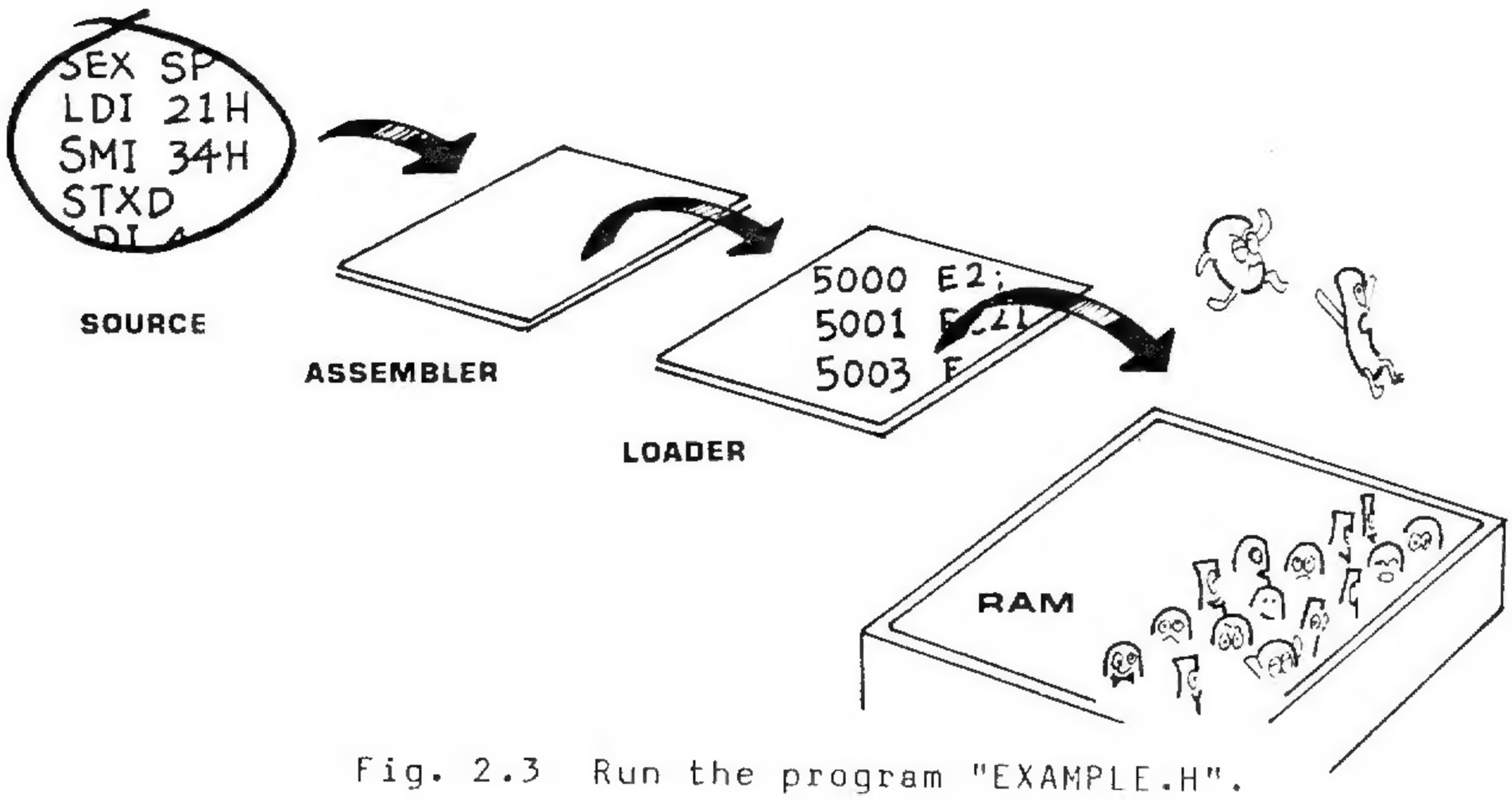

Fig. 2.3 Run the program "EXAMPLE.H".

COMX 35/PC-1

ASSEMBLER MANUAL

By : Charles Wong

Date : August 1, 1985

COMX World Operations Ltd.

R & D Department

Approved by : Edmond Leung

Table of Contents

CHAPTER 1

INTRODUCTION

1.1 General concept:

The COMX 18Ø2 Assembler is a two pass assembler. All the tables including the symbol table, pseudo instruction table(for MCASE, MIF, MLOOP) and directive table(for conditional assembling) are generated in pass one. If table or memory is full, the assembling will be aborted. According to the user's selected option, the hex code files and/or list files will be generated on disk, to printer or thermal printer and/or on screen.

1.2 How to call the Assembler?

After generated the text file to be assembled, the user can type: 'DOSURUN,"ASM2"', then the following information will be displayed on the screen.

```
SOURCE FILE (FN. , DR. NO. ; OPTIONS):
>
```

In the displayed statement, "FN." stands for source file name, "DR. NO." stands for disk drive number(1 or 2), "OPTIONS" stands for the options including G, H, W, D, X and L, which will be described in details later. The delimiter between two options is a space.

If the H option is selected, the following statement will be displayed, where "FN." stands for file name to be created for the hex. file, "DR. NO." stands for disk drive number. The default drive number is drive 1.

```
HEX. FILE (FN. , DR. NO.)
>
```

The following statement will be displayed after the above question or the first question (if H option was not selected) has been answered.

```
DESTINATION:
1 SC, 2 PR, 3 PR & SC, 4 TH, 5 TH & SC
6 DK, 7 DK & SC, 8 NO DEST

SC: SCREEN; PR: PRINTER; TH: THERMAL PRINTER; DK: DISK
NO DEST: NO DESTINATION
```

If 6 or 7 of the destinations is selected, another question which is shown below will be displayed on the screen, otherwise, the input text will be assembled.

```
DESTINATION FILE (FN. ,DR. NO):
>
```

The destination file name and disk drive number should be filled. If no drive number is filled, drive 1 will be assumed.

1.3 Options:

G: Generate all codes: Prints all lines of codes generated by pseudo-instructions and general instructions.

L: Prints out all lines assembled.

H: Generates hex. file

X: Prints out symbol table entries

W number: sets the max. Width (column number) per page in the listing.(Do not set the width in excess of 132, otherwise, the default value 8Ø will be assumed.

D number: Page Depth- Sets the number of lines per listing page including headings and blank lines.

CHAPTER 2

SYSTEM ARCHITECTURE

2.1 Registers

CDP1802 register summary:

D	8	Bits	D Register (Accumulator)
DF	1	Bit	Data Flag (ALU Carry)
R	16	Bits	1 to 16 Scratchpad Registers
P	4	Bits	Designates which register is Program Counter
X	4	Bits	Designates which register is Data Pointer
N	4	Bits	Low-order Instruction Nibble
I	4	Bits	High-order Instruction Nibble
T	8	Bits	Holds old X, P after Interrupt
IE	1	Bit	Interrupt Enable
Q	1	Bit	Output Flip-flop

Interrupt Action: X and P are stored in T after executing current instruction; designator P is set to 1; designator X is set to 2; interrupt enable is reset to Ø (inhibit); and the interrupt request is serviced.

DMA Action: Finish executing current instruction. R(Ø) points to memory area for data transfer; data is loaded into or read out of memory; and increment R (Ø).

Note: In the event of concurrent DMA and INTERRUPT requests, DMA has priority.

External Flags: Four one-bit Flags set externally and tested by some branching instructions.

2.2 Registers used by the operating system and Assembler

Register Ø : contains the vector address of DMA.

Register 1 : contains the vector address of Interrupt request.

Register 2 : is used as stack pointer by the system in general case. The stack pointer can also be redefined by the user.

Register 3 : is used as program counter by the system in general case. The PC can also be redefined by the user.

Register 4 : contains the entry address of the routine for the Assembler instruction "CALL".

Register 5 : contains the entry address of the routine for the Assembler instrucion "EXIT".

Register 6 : contains the returning address of the current executing subroutine.

Register 7 to Register 15 can be used by the users.

CHAPTER 3

TEXT INPUT FORMAT

3.1 INTRODUCTION

Before using the assembler, the designated text source file must have been generated. The input format must obey some rules specified by the assembler.

This chapter will cover the input formats of the constant, operands(expressions), output port and label system etc..

3.2 Assembler Input Format

Each line is a text string terminated by an end-of-line (return) character. The line can have one to five fields:

1. an optional label field

2. an operation field

3. an operand field for some operations

4. an operation delimiter(;), then follows another 2 & 3.

NOTE: 2, 3, 4 can be repeated in one text line.

5. an optional comment field
   (: as the prefix of the comment)

There are also two special cases: if the first character of a line is a colon, the entire line will be considered to be a comment which is printed in the listing but will not be otherwise processed. If the line contains only an an end-of-line character, it will be ignored and a blank line is printed on the listing. The maximum length of an input line is 80 characters. Any character after the 80th character will be ignored.

3.3 The Label Field

The label field begins at the first character position of the line. Labels can be either optional (instructions) or required (EQU). Lines with a space in the first character position are considered to have no label.

The label must be a legal symbol name consisting of one to six uppercase or lowercase characters, decimal digits, or the characters "_" or ".", however the first character must be a letter. Labels (and names in general) must be unique, ie., they cannot be defined more than once.

Label is assigned a 16-bit value depending on the line's operation field. Instructions and most constant-definition operations cause the label to have the value of the program address of the first byte generated for the line. Labels on EQU operation is given the value of the result of evaluation of the operand field. Label on 'DS' operation is normally given the value of the program address counter.

3.4 Operation Field

This field follows the label field and must be separated by one or more spaces. It must contain the mnemonic name of 1802 instructions, a "pseudo-instruction", or an assembler directive. The assembler accepts instructions in uppercase or lowercase characters.

The pseudo-instructions also generate code.

3.5 Operand Field

The operand field follows, and must be separated by at least one space from the operation field.

There are two kinds of operand, constant (number or predefined label) and expression (numbers and/or predefined symbols which is operated and given the result)

More than one instruction (operation+ operand) can be put in the same line, a delimiter ";" is needed to separated each instruction. Therefore operation field and operand field can be repeated in one line.

3.6 Comment Field

The last field of the source statement is the optional comment field. The contents of this field is not translated into object code, rather it is copied to the program listing. There must be a prefix ":" to start the comment field.

3.7 More about the operand field

Operands of many instructions and assembler directives can include numeric expressions in one or more places. The assembler can process expressions of almost any complexity using a format similar to algebraic notation used in programming language such as BASIC. All operands and operators used signed or unsigned 16-bit binary integers in the range of Ø to 65535 for unsigned numbers, or -32768 to +32767 for signed numbers. In some cases, expressions are expected to return a byte value of Ø to 255 (such as in 8-bit register instructions). Parentheses can be used to alter the natural order of evaluation.

3.7.1 Expression operands

Decimal number: optional minus sign and one to five digit.

Hexdecimal number: one to four hexdecimal characters (Ø-9 and A-F) followed by a character H the first character must be Ø to 9, if not a dummy Ø is needed.

Binary number: one to sixteen binary digits (Ø or 1) followed by a character B.

Octal number: one to 8 octal characters (Ø-7) followed by a character O.

Instruction counter: dollar sign "$" represents the current program instruction counter value.

Symbolic name: one to six characters: A-Z, a-z, Ø-9, . or _. However, the first character cannot be a digit . and _

3.7.2 ASSEMBLER Operators by order of evaluation

1.	* multiplication	/ division
2.	+ addition	- subtraction
3.	= equal < smaller > greater <> not equal	
	>= greater or equal	<= smaller or equal
4.	& logical AND	! logical OR
	# logical XOR	^ one's compliment

3.7.3 Symbolic Name

Name are defined when first used as a label on an instruction or directive statement. They must be defined exactly once in the program. If a name is redefined (used as label more than once) an error message is printed on every definition.

Symbolic name are stored with their associated type and value in an assembler data structure called "symbol table", which occupies most of the assembler's memory space. About 10K of memory is used as symbol table. Each entry in the table requires 9 bytes, so up to 1130 names can be used in one assembly program.

CHAPTER 4

MNEMONICS

4.1 Control Instructions

op code	syntax		name	action
Ø0	IDL		IDLE	WAIT FOR DMA OR INTERRUPT; M(R(Ø))-->BUS
C4	NOP		NO OPERATION	CONTINUE
DN	SEP	reg	SET P	N-->P
EN	SEX	reg	SET X	N-->X
7B	SEQ		SET Q	1-->Q
7A	REQ		RESET Q	Ø-->Q
78	SAV		SAVE	T-->M(R(X))
79	MARK		PUSH X, P TO STACK	(X,P)-->T; (X,P)-->M(R(2)); THEN P-->X;R(2)-1
70	RET		RETURN	M(R(X))-->(X,P); R(X)+1;1-->1E
71	DIS		DISABLE	M(R(X))-->(X,P); R(X)+1.Ø-->1E

4.2 Memory Reference

OP Code	Syntax		Name	Action
ON	LDN	reg	LOAD VIAN	M(R(N))-->D;FOR N NOT Ø
4N	LDA	reg	LOAD ADVANCE	M(R(N))-->D;R(N)+1
FØ	LDX		LOAD VIAX	M(R(X))-->D
72	LDXA		LOAD VIAX AND ADVANCE	M(R(X))-->D;R(X)+1
F8	LDI	expr	LOAD IMMEDIATE	M(R(P))-->D;R(P)+1
5N	STR	reg	STORE VIAN	D-->M(R(N))
73	STXD		STORE VIA X AND DECREMENT	D-->M(RX));R(X)-1

4.3 Register Operations

OP Code	Syntax		Name	Action
1N	INC	reg	INCREMENT REG N	R(N)+1
2N	DEC	reg	DECREMENT REG N	R(N)-1
60	IRX		INCREMENT REG X	R(X)+1
8N	GLO	reg	GET LOW REG N	R(N).0-->D
AN	PLO	reg	PUT LOW REG N	D-->R(N).0
9N	GHI	reg	GET HIGH REG N	R(N).1-->D
BN	PHI	reg	PUT HIGH REG N	D-->R(N).1

4.4 Logic Operations**

OP Code	Syntax		Name	Action
F1	OR		OR	M(R(X)) OR D-->D
F9	ORI	expr	OR IMMEDIATE	M(R(P)) OR D-->;R(P)+1
F3	XOR		EXCLUSIVE OR	M(R(P)) XOR D-->D
FB	XRI	expr	EXCLUSIVE OR IMMEDIATE	M(R(P)) XOR D-->D;R(P)+1
F2	AND		AND	M(R(X)) AND D-->D
FA	ANI	expr	AND IMMEDIATE	M(R(P)) AND D-->D;R(P)+1
F6	SHR		SHIFT RIGHT	SHIFT D RIGHT, LSB(D)-->DF,0-->MSB(D)
76	.SHRC		SHIFT RIGHT WITH CARRY	SHIFT D RIGHT, LSB(D)-->DF,DF-->MSB(D)
	.RSHR		RING SHIFT RIGHT	
FE	SHL		SHIFT LEFT	SHIFT D LEFT, MSB(D)-->DF,0-->LSB(D)
7E	.SHLC		SHIFT LEFT WITH CARRY	SHIFT D LEFT, MSB(D)-->DF,DF-->LSB(D)
	.RSHL		RING SHIFT LEFT	SHIFT D LEFT, MSB(D)-->DF,DF-->LSB(D)

4.5 Arithmetic Operations**

OP Code	Syntax		Name	Action
F4	ADD		ADD	M(R(X))+D-->DF,D
FC	ADI	expr	ADD IMMEDIATE	M(R(P))+D-->DF,D;R(P)+1
74	ADC		ADD WITH CARRY	M(R(X))+D+DF-->DF,D
7C	ADCI	expr	ADD WITH CARRY, IMMEDIATE	M(R(P))+D+DF-->DF,D; R(P)+1
F5	SD		SUBTRACT D	M(R(X))-D-->DF,D

FD	SDI	expr	SUBTRACT D IMMEDIATE	M(R(P))-D-->DF,D; R(P)+1
75	SDB		SUBTRACT D WITH BURROW	M(R(X))-D-(NOT DF)-->DF, D;R(P)+1
7D	SDBI	expr	SUBTRACT WITH BURROW IMMEDIATE	M(R(P))-D-(NOT DF)-->DF, D;R(P)+1
F7	SM		SUBTRACT MEMORY	D-M(R(X))-->DF,D
FF	SMI	expr	SUBTRACT MEMORY IMMEDIATE	D-M(R(P))-->DF,D; R(P)+1
77	SMB		SUBTRACT MEMORY WITH BURROW	D-M(R(X))-(NOT DF)-->DF,D
7F	SMBI	expr	SUBTRACT MEMORY BPRROW IMMEDIATE	D-M(R(P))-(NOT DF)-->DF, D;R(P)+1

4.6 Short Branch

OP Code	Syntax		Name	Action
3Ø	BR	expr	SHORT BRANCH	M(R(P))-->R(P)Ø
38	.NBR	expr	NO SHORT BRANCH (SEE SKP)	R(P)+1
32	BZ	expr	SHORT BRANCH IF D=Ø	IF D=Ø,M(R(P))Ø ELSE R(P)+1
3A	BNZ	expr	SHORT BRANCH IF D NOT Ø	IF D NOT Ø.M(R(P))--> R(P) Ø ELSE R(P)+1
33	.BDF	expr	SHORT BRANCH IF	IF DF=1,M(R(P))-->(P)Ø
	.BPZ	expr	SHORT BRANCH IF POS OR ZERO	ELSE R(P)+1
	.BGE	expr	SHORT BRANCH IF GREATER OR EQUAL	
3B	.BNF	expr	SHORT BRANCH IF	IF DF=Ø,M(R(P))-->R(P) Ø
	.BM	expr	SHORT BRANCH IF MINUS	ELSE R(P)+1
	.BL	expr	SHORT BRANCH IF LESS	
31	.BQ	expr	SHORT BRANCH IF Q=1	IF Q=1,M(R(P))-->R(P) Ø ELSE R(P)+1
39	.BNQ	expr	SHORT BRANCH IF Q=Ø	IF Q=Ø,M(R(P)) Ø ELSE R(P)+1
34	.B1	expr	SHORT BRANCH IF EF1=1	IF EF1=1,M(R(P))-->R(P) Ø ELSE R(P)+1
3C	BN1	expr	SHORT BRANCH IF EF1=Ø	IF DF1=Ø,M(R(P)) Ø ELSE R(P)+1
35	B2	expr	SHORT BRANCH IF EF2=1	IF EF2=1,M(R(P))-->R(P) Ø ELSE R(P)+1
3D	BN2	expr	SHORT BRANCH IF EF2=Ø	IF EF2=Ø,M(R(P))-->R(P) Ø ELSE R(P)+1

36	B3	expr	SHORT BRANCH IF EF3=1	IF EF3=1,M(R(P))-->R(P) Ø ELSE R(P)+1
3E	BN3	expr	SHORT BRANCH IF EF3=Ø	IF EF3=Ø,M(R(P))-->R(P) Ø ELSE R(P)+1
37	B4	expr	SHORT BRANCH IF EF4=1	IF EF4=1,M(R(P))-->R(P) Ø ELSE R(P)+1
3F	BN4	expr	SHORT BRANCH IF EF4=Ø	IF EF4=Ø,M(R(P))-->R(P) Ø ELSE R(P)+1

4.7 Long Branch

OP Code	Syntax		Name	Action
CØ	LBR	expr	LONG BRANCH	M(R(P))-->R(P).1; M(R(P)+1)-->R(P).Ø
C8	.NLBR	expr	NO LONG BRANCH (SEE LSKP)	R(P)+2
C2	LBZ	expr	LONG BRANCH IF D=Ø	IF D=Ø,M(R(P))-->R(P).1; M(R(P)+1)-->R(P).Ø; ELSE R(P)+2
CA	LBNZ	expr	LONG BRANCH IF D NOT Ø	IF D NOT Ø,M(R(P))--> R(P).1;M(R(P)+1)-->R(P).Ø ELSE R(P)+2
C3	LBDF	expr	LONG BRANCH IF DF=1	IF DF=1,M(R(P))-->R(P).1; M(R(P)+1)-->R(P).Ø; ELSE R(P)+2
CB	LBNF	expr	LONG BRANCH IF DF=Ø	IF DF=Ø,M(R(P))-->R(P).1; ELSE R(P)+2
C1	LBNQ	expr	LONG BRANCH IF Q=1	IF Q=1,M(R(P))-->R(P).1; M(R(P)+1)-->R(P).Ø; ELSE R(P)+2
C9	LBNQ	expr	LONG BRANCH IF Q=Ø	IF Q=Ø,M(R(P))-->R(P).1; M(R(P)+1)-->R(P).Ø; ELSE R(P)+2

4.8 Skip Instructions

OP Code	Syntax	Name	Action
38	.SKP	SHORT SKIP (SEE NBR)	R(P)+1
C8	.LSKP	LONG SKIP (SEE NLBR)	R(P)+2
CE	LSZ	LONG SKIP IF D=Ø	IF D=Ø,R(P)+2;ELSE CONTINUE
C6	LSNZ	LONG SKIP IF D NOT Ø	IF D NOT Ø,R(P)+2;ELSE CONTINUE

CF	LSDF	LONG SKIP IF DF=1	IF DF=1,R(P)+2;ELSE CONTINUE
C7	LSNF	LONG SKIP IF DF=Ø	IF DF=Ø,R(P)+2;ELSE CONTINUE
CD	LSQ	LONG SKIP IF Q=1	IF Q=1,R(P)+2;ELSE CONTINUE
C5	LSNQ	LONG SKIP IF Q=Ø	IF Q=Ø,R(P)+2;ELSE CONTINUE
CC	LSIE	LONG SKIP IF IE=1	IF IE=1,R(P)+2;ELSE CONTINUE

4.9 Input-Output Byte Transfer

OP Code	Syntax		Name	Action
6N	OUT	dev	OUTPUT	M(R(X))-->BUS;R(X)+1; FOR N=1 TO 7
6N	INP	dev	INPUT	BUS-->M(R(X));BUS-->D; FOR N=9 TO F

NOTES:

N A HEX DIGIT

reg A HEX DIGIT,"R" FOLLOWED BY A HEX DIGIT, OR A SYMBOLIC NAME.

dev "1" THROUGH "7" OR A SYMBOLIC NAME IN THAT RANGE.

expr A CONSTANT,".", OR A SYMBOLIC NAME POSSIBLY PLUS ("+") OR MINUS ("-") A CONSTANT.

. THIS INSTRUCTION IS ASSOCIATED WITH MORE THAN ONE MNEMONIC.
EACH MNEMONIC IS INDIVIDUALLY LISTED.

.. THE ARITHMETIC OPERATIONS AND THE SHIFT INSTRUCTIONS ARE THE ONLY INSTRUCTIONS THAT CAN ALTER THE DF.
AFTER AN ADD INSTRUCTION;
DF=1 DENOTES A CARRY HAS OCCURRED
DF=Ø DENOTES A CARRY HAS NOT OCCURRED
AFTER A SUBTRACT INSTRUCTION:
DF=1 DEOTES NO BORROW;D IS A TRUE POSITIVE NUMBER
DF=Ø DENOTES A BORROW;D IS TWO'S COMPLEMENT
THE SYNTAX - (NOT DF) DENOTES THE SUBTRACTION OF THE BURROW

CHAPTER 5

PSEUDO INSTRUCTIONS

5.1 INTRODUCTION

"Pseudo-instructions" are special assembler statements that generate object code but do not correspond to actual CDP 1802 machine instructions. Their primary purpose is to create special sequences of constant data to be included in the program. Labels are optional on pseudo-instructions. The other special purpose of pseudo-instructions in this assembler is to generate some machine codes to do 16-bit operations and macro-nesting operations.

5.2 DC and DW Statements

The Declare Constant and Declare Word pseudo-instructions generate sequences of single (DC) and double (DW) constants within the program. The operand is a list of one or more expressions which are evaluated and output as constants. If more than one constant is to be generated, the expressions are separated by commas. DC will truncate the high byte if any of its expressions is a value of more than 255. If DW evaluates an expression with a value of less than 256, the high order-byte will be zero.

Example:

```
dc 1,20,40h
DC 5,01010101B,30H
DW 30,2000H,99H
dw 5,0101010101010101b,50h
```

5.3 DM Statement

This psuedo-instruction generate a series of bytes corresponding to a string of characters given as operand. The output bytes are the literal numeric value of each ASCII character in the string.

The operand string must be enclosed by delimters before the first character and after the last character. The characters that may be used for delimiters are: / and ". Both delimiters must be the same character and cannot be included in the string itself.

Example:

```
DM / you are programmers./
dm " I WRITE A PROGRAM."
```

5.4 Pseudo-instructions that do 16-bit operations

LDR reg1,reg2: load data M(R(reg1)) into reg2
content of reg1 unchanged

LDRA reg1,reg2: load data M(R(reg1)) into reg2
content of reg1 increased by 2

LDRI data,reg: load immediate data into reg;
where data can be an expression

STD reg1,reg2: store data in reg1 into M(R(reg2));
content of reg2 unchanged

STDA reg1,reg2: store data in reg1 into M(R(reg2));
content of reg2 increased by 2

STDAI data,reg: store immediate data into M(R(reg));
content of reg increased by 2

STDI data,reg: store immediate data into M(R(reg));
content of reg unchanged

TFR reg1,reg2: transfer content in reg1 to reg2

5.5 CALL and EXIT instructions

In the procedure of system initialization, a vector address of the entry point for the program which performs "call subroutine" is put into R4. In the same manner, a vector address for "return from subroutine" is put into R5. Therefore, CALL is equivalent to SEP R4 and EXIT is equivalent to SEP R5.

5.6 PSHS and PULS instructions

Assume resgister 2 is stack pointer.

Format:

```
PSHS reg1,reg2...,regn
PULS reg1,reg2...,regn
```

PSHS : push registers onto stack. Stack pointer is updated & points to the next stack available stack location.

PULS : pull registers from stack. Stack pointer is updated & points to the next stack available stack location.

The order of pushing is from left to right, ie., push reg1 first, then reg2 etc. .

The order of pulling is from right to left, ie., pull back regn, and then regn-1 etc. .

5.7 Macro-instructions

Nesting is permitted in the following instructions.

5.7.1 MIFxx, MELSE and MENDIF Instructions

xx: Z, NZ, DF, NF

Z: D=Ø
NZ: D<>Ø
DF: DF=1
NF: DF=Ø

EXAMPLE:

```
MIFZ
     STR R4
MELSE
     STR R5
MENDIF
```

NOTE: If D=Ø, store (D) into R4, otherwise, store (D) into R5.

5.7.2 MCASE, MOF v1....., MENDOF and MENDCASE Instructions

Example:

```
MCASE
MOF 1      : COMPARE (D) WITH 1, IF TRUE,
           : NEXT STATEMENT , OTHERWISE,
           : STATEMENT AFTER FIRST MENDOF

  STR R3   : AFTER DOING THE INSTRUCTIONS
           : INSIDE THE MOF LOOP, DO
           : INSTRUCTIONS FOLLOW MENDCASE

MENDOF
MOF 4
  STR R5
MENDOF
MENDCASE
```

5.7.3 MLOOP, MLEAVE and MENDLOOP

EXAMPLE:

```
  MLOOP
     LDA R7
     STR R8
     DEC R9
     GLO R9
     BNZ $+5
MLEAVE
     LDXA
     STR R7
MENDLOOP
```

NOTE: MLEAVE must be located between MLOOP and MENDLOOP. The instuctions between MLOOP and MENDLOOP will be executed repeatedly until the MLEAVE is encountered.

CHAPTER 6

DIRECTIVES

6.1 Introduction

Assembler directive statements give the assembler information that affects the assembly process, but do not cause code to be generated.

6.2 Conditional Assembling

IF Statement

Sets the sense of conditional assembly. The operand follows the statement determines assembling condition. If the evaulated result of the operand is true, the current assembling condition is maintained, otherwise the subsequent statements will be skipped.

The operand can be any expression whose evaluated result is in boolean form (true or false).

ELSE Statement

Switches the sense of the current level of conditional assembly; if the assembler is processing statement, it will switch to sipping and vice versa.

ENDIF

Ends the conditional assembling of the nearest IF/ELSE or IF statement(s).

NOTE: IF/ELSE/ENDIF or IF/ENDIF statements may be nested.
They may not be with labels.

GO label

Whenever the GO statement is encountered, the subsequent statements will be skipped (not to be assembled). When the specified label is encountered, the previous assembling condition will be resumed.

6.3 DS, EQU, END, LIST, NOLIST, ORG and PAGE Statement

DS

Declares storage.
Number of storage declared depends on the evaluated result of its operand expression.

EQU

Assigns a value to a label by evaluating its operand expression. The label name must not have been used previously. EQU statements must have a label.

END

Indicates the end of the program. Its use is optional.

LIST

Tells the assembler to resume echoing the source code and machine code, and thus cancels the effect of the NOLIST statement.

NOLIST

Directs the assembler to cease echoing the source code and machine code to the listing.

PAGE

Changed the current program counter to the starting point of next page. For example, if the current pc=4030 (hex), after the PAGE statement is encountered, pc is changed to 4100 (hex.).

ORG operand

Assigns a value to program counter by evaluating its operand expression.

NOTE: DO NOT PLACE ANY INSTRUCTION ON THE SAME LINE THE PAGE OR ORG STATEMENT LOACTED, OTHERWISE, THERE MAY BE LOGICAL ERRORS IN THE PROCESS OF TRANSFORMING HEX FILE TO OBJECT FILE.

CHAPTER 7

ASSEMBLER LISTING AND ERROR MESSAGES

7.1 The assembler listing format, by starting column, is

Column 1 to column 4 -- Address Field: displays the initial value of the program counter.

Column 6 to column 18-- Machine code field: any code generated by the statement is listed in this field. At most 6 bytes of machine code can be listed in one line. If G option has been selected and more than 6 bytes of machine code have been geanerated, the code will be listed on the next line.

Column 20 --Information field: the assembler display statement status by printing a one-character symbol in this column.

i.e.

W-- warning: generated for label too long; or long branches whose destination can be reached by a short branch; or input text line too long (more than 80 characters in one line).

E-- error: indicates statements that have one or more errors. Overrides the "W" symbols.

Column 21 to 24--Source line number: this is a sequence number assigned by the assembler to each source statement read, even if the line was not actually printed.

Column 25 -- input field: the text read is printed out in this field.

If the line length is longer than 80 character (or any value given in an option W statement), the line will be truncated on the right.

7.2 Error Message

The assembler will print a brief error message for each error it detects. The message are printed before the line in error. Here is a summary of the assembler error messages.

1. BAD INSTRUCTION: the assembler did not recognize the instruction mnemonic.

2. LABEL MISSING: A label is needed in the line.

3. PHASING ERROR: The label value which was generated in pass 1 is different pass 2.

4. INVADID LABEL: The statement's label included an illegal character or did not start with a letter.

5. REDEFINED NAME: The used as label more than once.

6. REGISTER ERROR: The operand expression which represented a register number was at fault.

7. SYNTAX ERROR: Improper use of the nesting pseudo-instruction.

8. >255: The result of the expression was too large to fit in the required byte.

9. OUT OF RANGE: The destination of the branch is too far to use a short branch.

10. DELIMITER ERROR: Inproper use of delimiter for DM instruction or no delimiters used.

11. I/O PORT NUMBER: Operand for OUT & INP is at fault.

12. EXPRESSION ERROR: Error in the operand field was encourted.

13. LOOPING ERROR: Error found in the uses of MLOOP, MLEAVE and MENDLOOP.

14. MIF ERROR: Error found in the uses of MIF, MELSE and MENDIF.

15. MCASE ERROR: Error found in the uses of MCASE, MOF, MENDOF and MENDCASE.

16. CONDITIONAL ASSEMBLE NODE TABLE FULL: Too much nesting for the conditional assembling.

17. CONDITIONAL ASSEMBLER STACK FULL: Too much nesting.

18. NESTING PSEUDO NODE TABLE FULL: Too much nesting.

19. NESTING PSEUDO STACK FULL

20. CONDITIONAL ASSEMBLER NESTING ERROR

21. NO SUCH DEVICE: The device selected as the output device does not exist.

22. DISK FULL: DISK MEDIUM FULL

23. INPUT LINE TOO LONG(TEXT INPUT)

24. INVALID DESTINATION NUMBER

25. INVALID NAME: Invalid file name has been input

26. NAME TOO LONG: Label name too long(in excess of 6 characters)

27. WRONG INPUT TYPE: Input wrong information or wrong input format in answering the questions displayed on screen before the assembling can be proceeded.

28. OPTIONS ERROR: Input the wrong information in answering which options is selected.

29. WIDTH TOO LARGE: Width set by users greater than 132

30. INVALID NUMBER: Destination number error.

31. DIGIT BUFFER FULL, DIGIT TOO LARGE

32. NUMBER OVERFLOW

33. SYMBOL TABLE FULL: Too much label, the program is aborted.

34. EXPRESSION ERR: MISSING OPERATOR(S)

35. EXPRESSION ERR: MISSING BRACKET

36. EXPRESSION ERR: MISSING OPERAND(S)

37. EXPRESSION ERR: CANNOT BE DIVIDED BY ZERO

38. EXPRESSION ERR: INVALID SYMBOL

39. EXPRESSION ERR: UNDEFINED SYMBOL

40. EXPRESSION ERR: INVALID CONSTANT

41. EXPRESSION ERR: SYMBOL MISSING

APPENDIX A

ASCII CODE TABLE

MOST SIGNIFICANT HEX DIGIT

	Ø	1	2	3	4	5	6	7
Ø	NUL	DLE	SP	Ø	@	P	\	P
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	/	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	↑	n	~
F	SI	US	/	?	O	←	o	DEL

NOTES :

(1) Parity bit in most significant hex digit not included.

(2) Characters in columns Ø and 1 (as well as SP and DEL) are non-printing.

(3) Model 33 Teletypewriter prints codes in columns 6 and 7 as if they were column 4 and 5 codes.

APPENDIX B

Example Assembly Progam Listing

```
COMX 35/PC-1 ASSEMBLER   V1.0                                  PAGE 001

0000 ;              0001 :TWO BYTES SUBTRACTION
0000 ;              0002 :SUBTRACT $1234 FROM $4321
0000 ;              0003 :RESULT PRINT ON SCREEN
0000 ;              0004
0000 ;              0005 ::: REGISTER LABEL
0002 ;              0006 SP      EQU 2
0000 ;              0007
0000 ;              0008 ::: SUBROUTINE LABEL
320F ;              0009 OUTPUT EQU 320FH
0000 ;              0010
0000 ;              0011 ::: MAIN PROGRAM
0000 ;              0012         ORG 5000H
5000 E2;            0013 START   SEX SP
5001 F821;          0014         LDI 21H
5003 FF34;          0015         SMI 34H          :SUBTRACT LOW BYTE
5005 73;            0016         STXD             :PUSH LOW RESULT
5006 F843;          0017         LDI 43H
5008 7F12;          0018         SMBI 12H         :SUBTRACT HIGH BYTE
500A D4320F;        0019         CALL OUTPUT      :PRINT RESULT HIGH BYTE
500D 60F0;          0020         IRX ; LDX        :PULL LOW RESULT
500F D4320F;        0021         CALL OUTPUT      :PRINT RESULT LOW BYTE
5012 D5;            0022         EXIT             :RETURN TO BASIC
5013 ;              0023         END

00000 ERROR(S)
00000 WARNING(S)
$0013 00019 PROGRAM BYTES GENERATED
$001B 0027 BYTES USED FOR SYMBOL
```

COMX LOADER MANUAL

By : Stanley Mak
Date : August 13, 1985
R & D dept.,
COMX World Operations Ltd.
Approved by : Edmond Leung

TABLE OF CONTENT

1. INTRODUCTION

COMX Loader is a ultility which convert a text file into a assembly file or program. Source file is the HEX file generated by COMX 1802 Assembler or file with the same format. Destination may be in form of machine codes filled in RAM or assembly file written to disk.

COMX Loader occupied the memory from $5000 to $B2FF. $5000-$9FFF is the write buffer. $A000-$A8FF is the read buffer. Main program starts at $A900. If disk is selected as destination, write buffer will be used. All data from $5000 to $B2FF will be destroyed. If RAM is selected as destination, write buffer will not be used. Memory from $4400 to $9FFF can be used by user.

COMX Loader can be start run by DOSURUN the program "LOADER" or CALL(▮A900) after "LOADER" has been loaded.

2. OPERATION

Insert a disk containing the program "LOADER" into disk drive and type "DOSURUN LOADER". The following text will be displayed :

```
COMX LOADER VERSION 1.0
COMX WORLD OPERATIONS LTD., 1985 (C)

SOURCE FILE NAME (FN,DR):
>
```

Key in file name of the source HEX file followed by a comma and drive number. If no drive number is defined, drive 1 will be selected. If the file does not exist, an error message

```
**** FILE NOT FOUND
```

will be displayed and another file name should be entered.

If the file exists, another text will be displayed :

```
SELECT DESTINATION TYPE:
1. DISK  2.RAM  3.RAM & DISK
>
```

Key in a number to select destination type.

Difference among the destination types are as follow:

Type 1 : The generated codes will be written to disk and an ASM file will be formed. The formed file will be an ordinary assembly file. It can be loaded and executed by DOS command "DOSLOAD" and "DOSURUN".

Type 2 : The generated codes will be directly filled into RAM. No disk file will be formed.

Type 3 : The generated codes will first be filled into RAM and then saved to disk. An ASM file is formed.

If destination type 1 or 3 is selected, the following message will be displayed:

```
DESTINATION FILE NAME (FN,DR):
>
```

Key in the file name of the ASM file to be formed followed by a comma and the drive number. If no drive number is defined, drive 1 will be selected. If the file name alrady exists in the disk, an error message

```
**** FILE ALREADY EXISTS
```

will appear and another file name should be entered.

If destination type 1 is selected, the following message will be display :

```
PROGRAM GAP BYTE (DEFAULT $FF):
>
```

Key in the hexadecimal gap byte code or hit "CR" for $FF. This gap byte is used to fill the program gaps (if any) appeared in the program.

The convertion process will now start. If any error occured during the process, error messages will be displayed (see appendix C). If no error message is displayed, it means that the convertion process has finished successfully.

However, if the program does not start at page boundary, a warning message will be displayed.

If destination type 1 is selected, the warning message will be as follow:

```
WARNING
PROGRAM NOT START AT PAGE BOUNDARY.
FILE FRONT($XX00-$XXYY) FILLED WITH $C4.
```

$XXYY is the program starting address and $XX00 is the page boundary address just in front of the program. This warning informs that COMX Loader has assigned the destination file to start at page boundary ($XX00). Locations between the page boundary and the program start has been filled with $C4.

If destination type 3 is selected, the warning message will be as follow :

```
PROGRAM NOT START AT PAGE BOUNDARY.
```

This warning informs that COMX Loader has start saved the destination file at the page boundary just in front of the program. Content of locations between the page boundary and the program start has also been saved to disk.

Note : During entering file names and destination type, a "CR" key instead of any input will abort the program.

3. WORKING PRINCIPLE

Source file will be read into the read buffer. Text in read buffer will be converted into machine codes. Afterwards, the codes will be handled according to different types of destination.

Destination type 1 :

The codes will be filled into the write buffer. If program gap appeared, "gap bytes" will be filled until next opcode appear. The "gap byte" code can be defined by the user. When the write buffer is full or source file is complete, the codes will be written to disk. Since the codes will not be filled into RAM,address range of the program is not limited. However, due to the sequential writting format, address overlapping (one instruction's address smaller than or equal to the previous instruction's address) is not allowed.

Destination type 2 :

The codes will be filled into RAM. They will be filled according to their addresses. Therefore, address overlapping is allowed and address range is limited to $4400-$9FFF. Program gaps will not be filled.

Destination type 3 :

The codes will be filled into RAM according their addresses. After all codes has been filled, they will be saved to disk. Similar to type 2, address range is limited to $4400-$9FFF and address overlapping is allowed. Program gaps will not be filled but the content in gaps will be saved to disk.

According to COMX DOS, the lower byte of the starting address of an assembly file must be $00. Therefore, if a program does not start at page boundary ($XX00) and destination type 1 or 3 has been selected, the following process will be done :

Destination type 1 :

Addresses between page boundary and program start will be filled with the instruction NOP ($C4). The program will be start saved from the page boundary. A warning message indicating the addresses which has been filled with NOP will be displayed.

Destination type 3 :

A warning message will be displayed. NOP will not be filled. The program will be start saved at the page

boundary in front of the program. In this case, the content of the addresses between the page boundary and the program start will also be saved.

APPENDIX A

Loading file example

This example demostrates how to load a file. Source to be loaded is a file called PROG.H in drive 1. Destination will be a ASM file called PROG.C in drive 2.

PROG.H is a HEX file generated by COMX 1802 Assembler :

```
5010 F80AB8;
5013 F800A8;
5016 F8FF;
5018 FF01;
501A 3A08;
501C 2888;
501E 3A06;
5020 98;
5021 3A06;
5023 D5
5024 ;
```

Example procedure :

Note : Words underlined stand for text input.
Words in small letters are explanations.

```
READY
:DOSURUN,"LOADER"                          start run COMX Loader
 ---------------
COMX LOADER VERSION 1.0
COMX WORLD OPERATIONS LTD., 1985 (C)

SOURCE FILE NAME (FN,DR):
>PROG.H,1                                  input source file name
 --------
SELECT DESTINATION TYPE:
1.DISK  2.RAM  3.RAM & DISK
>1                                         select disk as dest
 -
DESTINATION FILE NAME (FN,DR):
>PROG.C,2                                  input dest file name
 --------
WARNING                                    process completed
PROGRAM NOT START AT PAGE BOUNDARY.
FILE FRONT($5000-$500F) FILLED WITH $C4.


READY
:DOSLOAD,"PROG.C"/2                        load the dest file
 ----------------
```

PROG.C is loaded into the memory $5000-$5023 and the content is shown below :

```
5000  C4 C4 C4 C4 C4 C4 C4 C4 C4 C4 C4 C4 C4 C4 C4 C4
5010  F8 0A B8 F8 00 A8 F8 FF FF 01 3A 08 28 88 3A 06
5020  98 3A 06 D5 00 00 00 00 00 00 00 00 00 00 00 00
```

APPENDIX B

Merging file example

This example demonstrates how to merge three programs, PROG1, PROG2 and PROG3, into a program called CODE.

PROG1, PROG2 and PROG3 are three HEX files :

PROG1 :
```
5000 F861B8;
5003 F800A8;
5006 08;
5007 D4320F;
500A D45100;
500D D5;
500E ;
6100 0B;
6101 ;
```

PROG2 :
```
5100 E2;
5101 F80A73;
5104 D4320F;
5107 D42E42;
510A D46000;
510D 60F0;
510F FF0173;
5112 3A04;
5114 D5;
5115 ;
```

PROG3 :
```
6000 F80AB8;
6003 F800A8;
6006 F8FF;
6008 FF01;
600A 3A08;
600C 2888;
600E 3A06;
6010 98;
6011 3A06;
6013 D5;
6014 ;
```

Use COMX Loader to load these files separately. Select destination type 2 so that the programs can be loaded into RAM. From the programs, the lowest and highest addresses can be found as $5000 and $6100. Type DOSSAVE,"CODE",■5000,■6100 to save the programs. The program CODE will now contain the programs PROG1,PROG2 and PROG3.

Example procedure :

Note : Words underlined stand for text input.
Words in small letter are explanations.

```
READY
:DOSURUN,"LOADER"                          start run COMX Loader
 ----------------
COMX LOADER VERSION 1.0
COMX WORLD OPERATIONS LTD., 1985 (C)

SOURCE FILE NAME (FN,DR):
>PROG1,1                                   load first file
 -------
SELECT DESTINATION TYPE:
1.DISK  2.RAM  3.RAM & DISK
>2                                         select load in RAM
 -

READY
:DOSURUN,"LOADER"                          start run COMX Loader
 ----------------
COMX LOADER VERSION 1.0
```

```
COMX WORLD OPERATIONS LTD., 1985 (C)

SOURCE FILE NAME (FN,DR):
>PROG2,1                                  load second file
 -------
SELECT DESTINATION TYPE:
1.DISK  2. RAM  3. RAM & DISK
>2                                        select load in RAM
 -

READY
:DOSURUN,"LOADER"                         start run COMX Loader
 ----------------
COMX LOADER VERSION 1.0
COMX WORLD OPERATIONS LTD., 1985 (C)

SOURCE FILE NAME (FN,DR):
>PROG3,1                                  load third file
 -------
SELECT DESTINATION TYPE:
1.DISK  2.RAM  3.RAM & DISK
>2                                        select load in RAM
 -

READY
:DOSSAVE,"CODE",■5000,■6100               save merged program
 -------------------------
```

APPENDIX C

Error messages

1. FILE NAME TOO LONG	- File name exceeds 18 characters.
2. INCORRECT DRIVE NO.	- Drive number does not equal to 1 or 2.
3. FILE NOT FOUND	- Source file can not be found.
4. FILE ALREADY EXISTS	- Destination file name already exists.
5. SOURCE FILE NOT HEX FILE	- Format of source file not identical to COMX 1802 Assembler HEX file format.
6. ADDRESS OVERLAPPED	- Address overlapping occured.
7. ADDRESS NOT PERMITTED	- Address less then $4400 or greater then $9FFF.